ELSEVIER

Science of Computer Programming 31 (1998) 263–289

Science of Computer Programming

Structured Gamma

Pascal Fradet, Daniel Le Métayer*

IRISA/INRIA, Campus de Beaulieu, 35042 Rennes, France

Abstract

The Gamma language is based on the chemical reaction metaphor which has a number of benefits with respect to parallelism and program derivation. But the original definition of Gamma does not provide any facility for data structuring or for specifying particular control strategies. We address this issue by introducing a notion of *structured multiset* which is a set of addresses satisfying specific relations. The relations can be seen as a form of neighborhood between the molecules of the solution; they can be used in the reaction condition of a program or transformed by the action. A type is defined by a context-free graph grammar and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the grammar defining T. We define a type checking algorithm that allows us to prove mechanically that a program maintains its data structure invariant. We illustrate the significance of the approach for program refinement and we describe its application to coordination.

Keywords: Multiset rewriting; Graph grammar; Type checking; Program refinement; Coordination; Software architecture

1. Gamma: motivations and limitations

The fast evolution of hardware and the growing needs of end-users has placed new requirements on the design of programming languages: sequentiality should no longer be seen as the prime programming paradigm but just as one of the possible forms of cooperation between individual entities. The Gamma formalism was proposed ten years ago precisely to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A program is a pair (*Condition*, *Action*) called a reaction. Execution proceeds by replacing in the

* Corresponding author. E-mail: Daniel.lemetayer@irisa.fr.

0167-6423/98/$19.00
PII S0167-6423(97)00023-3

multiset elements satisfying the condition by the products of the action. The result is obtained when a stable state is reached, that is to say, when no more reactions can take place. The following is an example of a Gamma program computing the maximum element of a non-empty set:

$$max = [x,\ y,\ x \leqslant y \longmapsto y]$$

$x \leqslant y$ specifies a property to be satisfied by the selected elements x and y. These elements are replaced in the set by the value y. Nothing is said in this definition about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel. Let us consider, as another introductory example, a sorting program. We represent a sequence as a set of pairs (*index*,*value*) and the program exchanges ill-ordered values until a stable state is reached and all values are well-ordered.

$$sort = [(i,x),\ (j,y),\ (i<j),\ (x>y) \longmapsto (i,y),\ (j,x)]$$

The interested reader may find in [3] a longer series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [4] a review of contributions related to the chemical reaction model.

The possibility of getting rid of artificial sequentiality in Gamma has two important consequences:

- It confers a very high level nature to the language and allows the programmer to describe programs in a very abstract way. As a consequence, Gamma can be used as an intermediate language that makes it easier to derive a program from a specification by successive refinements [2].
- Because Gamma programs do not have any sequential bias, the language naturally leads to the construction of parallel programs (in fact, it is much harder to write a sequential program than a parallel program in Gamma). It is also suitable as the basis of a "coordination language" for the description of the overall interactions between individual entities in a large application [20].

However, our experience with Gamma also highlighted some weaknesses of the language. Let us now review the most important ones.

- The original definition of Gamma lacks any operation for combining programs.
- The language does not make it easy for the programmer to structure data or to specify particular control strategies.
- Because of the combinatorial explosion imposed by its semantics, it is difficult to reach a decent level of efficiency in any general purpose implementation of the language.

For the sake of modularity it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about programs. This issue was addressed in [13, 14] which introduce operators for the

parallel and the sequential composition of programs and study their properties and in [7, 19] which define higher-order extensions of Gamma. Another approach was taken in [6] where a notion of *schedules* is proposed to control the execution of Gamma programs.

The lack of support for structuring data and the difficulty of imposing a particular control strategy should not be surprising since the original motivation for the language was to be able to describe programs exhibiting as few ordering constraints as possible. An unfortunate consequence however is that the programmer sometimes has to resort to artificial encodings to express his algorithm. For instance, the exchange sort algorithm shown above is expressed in terms of multisets of pairs (*index*,*value*). This limitation also introduces an unnecessary factor of inefficiency in the implementation because the underlying structure of the data (and control) is not exposed to the compiler. Such information could be exploited to improve the implementation [9] but it can usually not be recovered by an automatic analysis of the program.

So, the lack of structuring facility is detrimental both for reasoning about programs and for implementing them. In this paper, we propose a solution to this problem without jeopardizing the basic qualities of the language. Let us point out in particular that it would not be acceptable to take the usual view of recursive type definitions because this would lead to a recursive style of programming and ruin the fundamental locality principle (because the data structure would then be manipulated as a whole). Our proposal is based on a notion of *structured multiset* which is a set of addresses satisfying specific relations and associated with values. The relations express a form of neighborhood between the molecules of the solution; they can be used in the reaction condition of a program or transformed by the action. In our framework, a type is defined in terms of rewrite rules on the relations of a multiset; a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. The paper defines a type checking algorithm that allows us to prove mechanically that a program maintains its data structure invariant. We illustrate the significance of the approach for program refinement and we describe its application to coordination.

We define the notion of structured multiset and structured program in Section 2. We describe the syntax and a formal semantics of this extension of Gamma and suggest how Structured Gamma programs can be translated in a straightforward way into original Gamma programs. The notion of structuring types is introduced in Section 3 with a collection of examples illustrating the programming style of Structured Gamma. In Section 4, we describe a checking algorithm and show its correctness. The correctness property is akin to the subject reduction property of type systems for functional languages. We illustrate the type system and type checking algorithm with several examples. Section 5 introduces the notions of type and program refinement which can be used to derive efficient implementations from Gamma specifications. Section 6 presents the application of structured multisets to coordination and the conclusion suggests avenues for further research.

2. Syntax and semantics of Structured Gamma

A structured multiset is a set of addresses satisfying specific relations. As an example, the list [5; 2; 7] can be represented by a structured multiset whose set of addresses is $\{a_1, a_2, a_3\}$ and associated values (written $\overline{a_i}$) are $\overline{a_1} = 5$, $\overline{a_2} = 2$, $\overline{a_3} = 7$. Let **next** be a binary relation and **end** a unary relation; the addresses satisfy

next a_1 a_2, **next** a_2 a_3, **end** a_3

A Structured Gamma program is defined in terms of pairs of a condition and an action which can

- test/modify the relations on addresses,
- test/modify the values associated with addresses.

As an illustration, an exchange sort for lists can be written in Structured Gamma as

$$Sort = [\textbf{next}\ x\ y,\ \overline{x} > \overline{y} \longmapsto \textbf{next}\ x\ y,\ x := \overline{y},\ y := \overline{x}]$$

The two selected addresses x and y must satisfy the relation **next** x y and their values $\overline{x}$ and $\overline{y}$ are such that $\overline{x} > \overline{y}$. The action exchanges their values and leaves the relation unchanged. To be complete, we should also declare that the multiset rewritten by *Sort* is of type *List* and that the reaction preserves the type of the multiset. These two points are treated, respectively, in Sections 3 and 4.

In order to define the syntax and semantics of Structured Gamma, we consider three basic domains:

- **R**: the set of relation symbols,
- **A**: the set of addresses,
- **V**: the set of values.

2.1. Syntax

The syntax of Structured Gamma programs is described by the following grammar:

$$\begin{aligned}
\langle Program\rangle &::= \text{ProgName} = [\langle Reaction\rangle]^* \\
\langle Reaction\rangle &::= \langle Condition\rangle \longmapsto \langle Action\rangle \\
\langle Condition\rangle &::= \mathbf{r}\, x_1 \cdots x_n \mid f^{Bool}(\overline{x_1}, \ldots, \overline{x_n}) \mid \\
&\qquad \langle Condition\rangle, \langle Condition\rangle \\
\langle Action\rangle &::= \mathbf{r}\, x_1 \cdots x_n \mid x := f^{\mathbf{V}}(\overline{x_1}, \ldots, \overline{x_n}) \mid \langle Action\rangle, \langle Action\rangle
\end{aligned}$$

where $\mathbf{r}$ ($\in \mathbf{R}$) denotes an n-ary relation, x_i is an address variable, $\overline{x_i}$ is the value at address x_i and $f^{\mathbf{X}}$ is a function from $\mathbf{V}^n$ to $\mathbf{X}$.

As can be seen in the *Sort* example, $\overline{x}$ always refers to the value of address x at selection time. This makes the evaluation order of the basic operations of an action (in particular, assignments) semantically irrelevant. In order to fit with this design choice, a Structured Gamma program must satisfy two additional syntactic conditions:

- If $\overline{x}$ occurs in the reaction then x occurs in the condition.
- An action may not include two assignments to the same variable.

2.2. Semantics

We write $\mathbf{A}(M)$ to denote the set of addresses occurring in the multiset M and "+" the multiset union. A structured multiset M can be seen as $M = Rel + Val$ where

- *Rel* is a *multiset* of relations represented as tuples $(\mathbf{r}, a_1, \ldots, a_n)$ (with $\mathbf{r} \in \mathbf{R}$ and $a_i \in \mathbf{A}$)
- *Val* is a *set* of values represented by triplets of the form $(\mathbf{val}, a, v)$ (with $a \in \mathbf{A}$ and $v \in \mathbf{V}$)

For example, the structured multiset shown at the beginning of this section can be written:

$$\{(\mathbf{next}, a_1, a_2), (\mathbf{next}, a_2, a_3), (\mathbf{end}, a_3), (\mathbf{val}, a_1, 5), (\mathbf{val}, a_2, 2), (\mathbf{val}, a_3, 7)\}$$

A valid structured multiset is such that an address x does not have more than one value (i.e. x occurs at most once in *Val*). On the other hand, there may be several occurrences of the same tuple in *Rel*. Also, we do not enforce that

$$\mathbf{A}(Rel) \subseteq \mathbf{A}(Val) \quad \text{nor that} \quad \mathbf{A}(Val) \subseteq \mathbf{A}(Rel)$$

So, addresses are allowed not to possess a value or may have a value without occurring in a relation. In the latter case however, they cannot be accessed by a Structured Gamma program and may be garbage collected.

In order to define the semantics of programs, we associate three functions with each reaction $C \models\!\!\Rightarrow A$. They are presented in Fig. 1.

The boolean function $\mathscr{T}(C)$ represents the condition of application of a reaction. The function $\mathscr{C}(C)$ represents the tuples selected by the condition (i.e. the relations and values occurring in C). The function $\mathscr{A}(A)$ represents the tuples added by the action, that is to say: the relations occurring in A, the values selected but unchanged by the reaction and the assigned values.

The semantics of a Structured Gamma program

$$P = [C_1 \models\!\!\Rightarrow A_1, \ldots, C_m \models\!\!\Rightarrow A_m]$$

applied to a multiset M is defined as the set of normal forms of the following rewrite system:

$$\begin{aligned} M &\longrightarrow_P \mathscr{G}\mathscr{C}(M) \\ &\quad \text{if} \quad \forall \{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M) \quad \forall i \in [1, \ldots, m] \ \neg \mathscr{T}(C_i)(x_1, \ldots, x_n) \\ M &\longrightarrow_P M - \mathscr{C}(C_i)(x_1, \ldots, x_n) + \mathscr{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k) \\ &\quad \text{with } y_1, \ldots, y_k \notin \mathbf{A}(M) \\ &\quad \text{and } \{x_1, \ldots, x_n\} \subseteq \mathbf{A}(M), \quad i \in [1, \ldots, m] \text{ and } \mathscr{T}(C_i)(x_1, \ldots, x_n) \end{aligned}$$

If no tuple of addresses satisfies any condition then a normal form is found. The result is the accessible structure described by the relations. The function $\mathscr{G}\mathscr{C}$ removes from *Val* the addresses not occurring in *Rel*. More formally:

$$\mathscr{G}\mathscr{C}(Rel + Val) = Rel + \{(\mathbf{val}, a, v) \mid (\mathbf{val}, a, v) \in Val \wedge a \in \mathbf{A}(Rel)\}$$

$$\mathcal{T}(C)(a_1,\cdots,a_i,b_1,\cdots,b_j) = (\mathbf{val},a_1,\overline{a_1}) \in Val \wedge\cdots\wedge (\mathbf{val},a_i,\overline{a_i}) \in Val$$
$$\wedge\ (\mathbf{val},b_1,\overline{b_1}) \in Val \wedge\cdots\wedge (\mathbf{val},b_j,\overline{b_j}) \in Val$$
$$\wedge\ \lfloor C \rfloor$$

$$\mathcal{C}(C)(a_1,\cdots,a_i,b_1,\cdots,b_j) = \{(\mathbf{val},a_1,\overline{a_1}),\cdots,(\mathbf{val},a_i,\overline{a_i}),$$
$$(\mathbf{val},b_1,\overline{b_1}),\cdots,(\mathbf{val},b_j,\overline{b_j})\} + \lceil C \rceil$$

$$\mathcal{A}(A)(a_1,\cdots,a_i,b_1,\cdots,b_j,c_1,\cdots,c_k) = \{(\mathbf{val},a_1,\overline{a_1}),\cdots,(\mathbf{val},a_i,\overline{a_i})\} + \lceil A \rceil$$

where $\lfloor\ \rfloor$ is defined by:

$$\lfloor X_1, X_2 \rfloor = \lfloor X_1 \rfloor \wedge \lfloor X_2 \rfloor$$
$$\lfloor \mathbf{r}\ x_1\cdots x_n \rfloor = (\mathbf{r},\ x_1,\cdots,x_n) \in Rel$$
$$\lfloor f(\overline{x_1},\cdots,\overline{x_n}) \rfloor = f(\overline{x_1},\cdots,\overline{x_n})$$

$\lceil\ \rceil$ is defined by:

$$\lceil X_1, X_2 \rceil = \lceil X_1 \rceil + \lceil X_2 \rceil$$
$$\lceil \mathbf{r}\ x_1\cdots x_n \rceil = \{(\mathbf{r},\ x_1,\cdots,x_n)\}$$
$$\lceil f(\overline{x_1},\cdots,\overline{x_n}) \rceil = \emptyset$$
$$\lceil x := f(\overline{x_1},\cdots,\overline{x_n}) \rceil = \{(\mathbf{val},x,f(\overline{x_1},\cdots,\overline{x_n}))\}$$

and

- $\{a_1,\cdots,a_i\}$ denotes the set of non-assigned variables whose value occurs in the reaction,
- $\{b_1,\cdots,b_j\}$ denotes the set of assigned variables occurring in the condition C,
- $\{c_1,\cdots,c_k\}$ denotes the set of variables occurring only in the action A.

Fig. 1. Semantic functions.

We use the notation $M \overset{*}{\longmapsto}_P M'$ for $M \overset{*}{\longrightarrow}_P M'$ and M' is a normal form for P.

Otherwise, a tuple of addresses $(x_1,\ldots,x_n)$ and a pair (C_i,A_i) such that $\mathcal{T}(C_i)(x_1,\ldots,x_n)$ are non-deterministically chosen. The multiset is transformed by removing $\mathcal{C}(C_i)(x_1,\ldots,x_n)$, allocating fresh addresses $y_1,\ldots,y_k$ and adding $\mathcal{A}(A_i)(x_1,\ldots,x_n, y_1,\ldots,y_k)$.

Note that the semantics enforces that different variable names in the program must be instantiated with different addresses. Sometimes, this requirement may lead to unnecessary verbose programs. For example, if we want to express the rewriting of any instance of a relation tuple $(\mathbf{r},x_1,\ldots,x_n)$ in A, we would like to write $\mathbf{r}\ x_1\cdots x_n \Longrightarrow A$ assuming x_i and x_j may possibly denote the same address rather than enumerating all the possible sharing patterns. Let us note, however, that it is always possible to translate the rule above into an equivalent set of rules where variables cannot be identified. So, a sensible option would be to address the matter at the syntax level and add a special notation to denote that some variables may be identified. For example, $\mathbf{r}\ x\_1\ y\_2\ z\_1\ t\_2$ would mean that x and z may be equal, y and t may be equal but x and z are different from y and t. This syntax could be automatically translated into standard rules.

2.3. Correspondence between Structured Gamma and original Gamma

Compared to the original Gamma formalism, the basic model of computation remains unchanged. It still consists in repeated applications of local actions in a global data structure. Actually, our way to define the semantics of Structured Gamma programs is very close to a translation into equivalent pure Gamma programs.

Rather than providing a formal definition of the translation, we illustrate it with the exchange sort program which is defined as follows in Structured Gamma:

$$Sort = [\mathbf{next}\ x\ y,\ \overline{x} > \overline{y} \longmapsto \mathbf{next}\ x\ y,\ x := \overline{y},\ y := \overline{x}]$$

and can be rewritten in pure Gamma as

$$Sort = [(\mathbf{val}, x, \overline{x}),\ (\mathbf{val}, y, \overline{y}),\ (\mathbf{next}, x, y),\ \overline{x} > \overline{y} \longmapsto (\mathbf{next}, x, y),\ (\mathbf{val}, x, \overline{y}),\ (\mathbf{val}, y, \overline{x})]$$

3. Structuring types

Structured multisets can be seen as a syntactic facility allowing the programmer to make the organization of the data explicit. We are now in a position to introduce a new notion of type which characterizes the structure of a multiset. We define a type in terms of rewrite rules on the relations of the multiset. A structured multiset is said to belong to a type if its underlying set of addresses can be produced by the rewrite system defining the type. We provide a formal definition of types and we illustrate them with a collection of examples.

3.1. Syntax

The syntax of types is defined by the following grammar:

$$\langle TypeDecl \rangle ::= \text{TypeName} = \langle Prod \rangle, [\langle NonTerm \rangle = \langle Prod \rangle]^*$$

$$\langle NonTerm \rangle ::= \text{NonTerminalName}\ x_1 \cdots x_n$$

$$\langle Prod \rangle ::= \mathbf{r}\ x_1 \cdots x_n \mid \langle NonTerm \rangle \mid \langle Prod \rangle, \langle Prod \rangle$$

where $\mathbf{r}$ ($\in \mathbf{R}$) is an n-ary relation ($n > 0$), and x_i is a variable denoting an address.

A type definition resembles a context-free graph grammar. For example, lists can be defined as

$$List = L\ x$$

$$L\ x = \mathbf{next}\ x\ y,\ L\ y$$

$$L\ x = \mathbf{end}\ x$$

3.2. Semantics

The definition of a type T can be associated with a Structured Gamma program (written Gen^T) which can return any multiset of type T. It amounts to considering '=' symbols as '$\Longmapsto$' and nonterminal names as relations. We keep the same notation $NTx_1 \cdots x_p$ to denote a nonterminal in a type definition or a relation in the rewrite system associated with a type. The correct interpretation is usually clear from the context. For example, the Structured Gamma program associated with the type *List* is defined by

$$\begin{aligned} Gen^{List} = [\mathit{List} &\Longmapsto L\,x \\ L\,x &\Longmapsto \textbf{next}\ x\ y,\ L\ y \\ L\,x &\Longmapsto \textbf{end}\ x] \end{aligned}$$

This program applied to a multiset containing only the atom *List* can produce all the finite lists.

We write $|M|$ to denote the multiset restricted to relations; formally:

$$|Rel + Val| = Rel$$

Definition 1. A multiset M has type T (written $M : T$) iff $\{T\} \overset{*}{\longmapsto}_{Gen^T} |M|$.

The inverse of Gen^T is a rewrite system (denoted by $\rightsquigarrow_T$) that provides a useful alternative definition of types. In this paper, reasoning about types is done using this rewrite system instead of *Gen*.

Definition 2. $M \longrightarrow_{Gen^T} M' \Leftrightarrow M' \rightsquigarrow_T M$

Proposition 3. *A multiset M has type T iff $|M| \overset{*}{\rightsquigarrow}_T \{T\}$.*

For example, the rewrite system associated with the type *List* is written as follows:

$$\begin{aligned} L\,x &\rightsquigarrow_{List} \mathit{List} \\ \textbf{next}\ x\ y,\ L\ y &\rightsquigarrow_{List} L\,x \\ \textbf{end}\ x &\rightsquigarrow_{List} L\,x \end{aligned}$$

This system rewrites any multiset of type *List* into the singleton $\{List\}$. Let us point out that $\rightsquigarrow_T$ reductions must enforce that if a variable of the *lhs* does not occur in the *rhs* then it does not occur in the rest of the multiset. For example, the rule **next** $x\ y,\ L\ y \rightsquigarrow_{List} L\,x$ cannot be applied to a multiset containing other occurrences of y. This requirement is dual to the constraint in the semantics of Structured Gamma programs that enforces variables of the *rhs* of a reaction not occurring in the *lhs* to be fresh. It is a global operation and such rewriting systems are clearly not Structured Gamma programs.

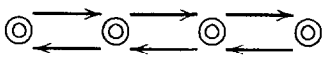
3.3. Examples of types

Abstract types found in functional languages such as ML can be defined in a natural way in Structured Gamma. For example, the type corresponding to binary trees is

$$\begin{aligned} \mathit{Bintree} &= B\,x \\ B\,x &= \textbf{node}\ x\ y\ z,\ B\ y,\ B\ z \\ B\,x &= \textbf{leaf}\ x \end{aligned}$$

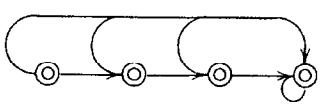
However, structuring types are expressive enough to describe not only tree shaped but also graph structures. Actually, the main blessing of the framework is to allow concise definitions of complicated pointer-like structures. To give a few examples, it is quite easy to define common pointer structures such as

Doubly-linked lists:



$$\begin{aligned} \mathit{Doubly} &= L\,x \\ L\,x &= \textbf{next}\ x\ y,\ \textbf{pred}\ y\ x,\ L\ y \\ L\,x &= \textbf{end}\ x \end{aligned}$$

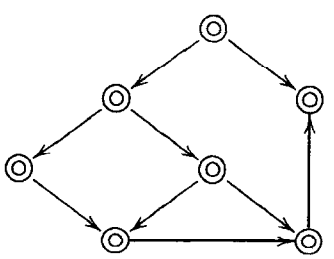
Lists with connections to the last element:



$$\begin{aligned} \mathit{Listlast} &= L\,x\,z \\ L\,x\,z &= \textbf{next}\ x\ y,\ \textbf{last}\ x\ z,\ L\ y\ z \\ L\,x\,z &= \textbf{next}\ x\ z,\ \textbf{last}\ x\ z,\ \textbf{next}\ z\ z \end{aligned}$$

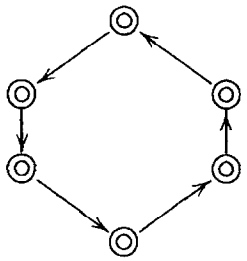
Binary trees with linked leaves:

$$\begin{aligned} \mathit{Binlink} &= L\,x\,y\,z \\ L\,x\,y\,z &= \textbf{left}\ x\ l,\ \textbf{right}\ x\ r,\ L\ l\ y\ u,\ L\ r\ v\ z,\ \textbf{next}\ u\ v \\ L\,x\,y\,z &= \textbf{left}\ x\ l,\ \textbf{right}\ x\ z,\ L\ l\ y\ u,\ \textbf{next}\ u\ z \\ L\,x\,y\,z &= \textbf{left}\ x\ y,\ \textbf{right}\ x\ r,\ L\ r\ v\ z,\ \textbf{next}\ y\ v \\ L\,x\,y\,z &= \textbf{left}\ x\ y,\ \textbf{right}\ x\ z,\ \textbf{next}\ y\ z \end{aligned}$$



The grammars can be explained by attaching a meaning to each nonterminal. For example, in the last example, the nonterminal $L\,x\,y\,z$ denotes a binary tree with linked leaves with root x, leftmost leaf y and rightmost leaf z.

Let us point out that the definition of a type T in terms of Gen^T implies that different variables denote different addresses in type definitions (as for program definitions). This choice entails the same drawbacks and calls for the same solution as in the case of programs. For example, using the notation hinted at in Section 2.2, circular lists can be defined by



$$\begin{aligned} \mathit{Circular} &= L\ x\ x \\ L\ x\_1\ y\_1 &= L\ x\ z,\ L\ z\ y \\ L\ x\_1\ y\_1 &= \mathbf{next}\ x\ y \end{aligned}$$

which is expanded into the following type in the pure Structured Gamma syntax:

$$\begin{aligned} \mathit{Circular} &= L\ x\ x \\ L\ x\ x &= L\ x\ z,\ L\ z\ x \\ L\ x\ x &= \mathbf{next}\ x\ x \\ L\ x\ y &= L\ x\ z,\ L\ z\ y \\ L\ x\ y &= \mathbf{next}\ x\ y \end{aligned}$$

3.4. *Programming using structuring types*

Many programs are expressed more naturally in Structured Gamma than in pure Gamma. The underlying structure of the multiset can be described by a type whereas in pure Gamma we had to encode it using tuples and tags. Let us give a few examples of Structured Gamma programs whose description in pure Gamma is cumbersome. Note that the syntax of programs is extended to account for typed programs (ProgName : TypeName = $\cdots$).

Iota takes a singleton $[\overline{a}]$ and yields the list $[\overline{a}; \overline{a-1}; \ldots; 1]$.

$$\mathit{Iota} : \mathit{List} = [\mathbf{end}\ x,\ \overline{x} > 1 \longmapsto \mathbf{next}\ x\ y,\ \mathbf{end}\ y,\ y := \overline{x} - 1]$$

MultB takes a binary tree representing an arithmetic expression and yields a leaf whose value is the evaluation of the original expression.

$$\begin{aligned} \mathit{MultB} : \mathit{Bintree} = [& \\ & \mathbf{node}\ x\ y\ z,\ \mathbf{leaf}\ y,\ \mathbf{leaf}\ z,\ \overline{x} = '+' \longmapsto \mathbf{leaf}\ x,\ x := \overline{y} + \overline{z} \\ & \mathbf{node}\ x\ y\ z,\ \mathbf{leaf}\ y,\ \mathbf{leaf}\ z,\ \overline{x} = '-' \longmapsto \mathbf{leaf}\ x,\ x := \overline{y} - \overline{z} \\ & \mathbf{node}\ x\ y\ z,\ \mathbf{leaf}\ y,\ \mathbf{leaf}\ z,\ \overline{x} = '*' \longmapsto \mathbf{leaf}\ x,\ x := \overline{y} * \overline{z} \\ & \mathbf{node}\ x\ y\ z,\ \mathbf{leaf}\ y,\ \mathbf{leaf}\ z,\ \overline{x} = '/' \longmapsto \mathbf{leaf}\ x,\ x := \overline{y} / \overline{z}] \end{aligned}$$

Types can also be used to express precise control constraints. For example, lists can be defined with two identified elements used as pointers to enforce a specific reduction strategy.

$$
\begin{aligned}
List_m &= L_0\ x \\
L_0\ x &= \mathbf{m1}\ x,\ \mathbf{next}\ x\ y,\ L_1\ y \\
L_0\ x &= \mathbf{next}\ x\ y,\ L_0\ y \\
L_1\ x &= \mathbf{m2}\ x,\ L_2\ x \\
L_1\ x &= \mathbf{next}\ x\ y,\ L_1\ y \\
L_2\ x &= \mathbf{next}\ x\ y,\ L_2\ y \\
L_2\ x &= \mathbf{end}\ x
\end{aligned}
$$

The type definition enforces that **m1** identifies a list element located before the element marked by **m2**. Assuming an initial list where **m1** marks the first element and **m2** the second one, we can describe a sequential sort.

$$
\begin{aligned}
&SeqSort : List_m = [\\
&\mathbf{m1}\ x,\ \mathbf{m2}\ y,\ \overline{x} > \overline{y} && \Longmapsto\ \mathbf{m1}\ x,\ \mathbf{m2}\ y,\ x := \overline{y},\ y := \overline{x} \\
&\mathbf{m1}\ x,\ \mathbf{m2}\ y,\ \mathbf{next}\ y\ z,\ \overline{x} \leqslant \overline{y} && \Longmapsto\ \mathbf{m1}\ x,\ \mathbf{m2}\ z,\ \mathbf{next}\ y\ z \\
&\mathbf{m1}\ x,\ \mathbf{m2}\ y,\ \mathbf{end}\ y,\ \mathbf{next}\ x\ z,\ && \Longmapsto\ \mathbf{m1}\ z,\ \mathbf{m2}\ w,\ \mathbf{end}\ y,\ \mathbf{next}\ x\ z, \\
&\mathbf{next}\ z\ w,\ \overline{x} \leqslant \overline{y} && \quad\ \mathbf{next}\ z\ w]
\end{aligned}
$$

In fact, $List_m$ can be shown more precisely to be a refinement of *List*. We come back to this issue in Section 5.

To summarize, Structured Gamma retains the spirit of Gamma while providing means to declare data structures and to enforce specific reduction strategies (e.g. for efficiency purposes). We describe in the following section another major benefit of Structured Gamma: the possibility for the programmer to have his programs checked to ensure that the data structure is manipulated in a consistent way.

4. Static type checking

The natural question following the introduction of a new type system concerns the design of an associated type checking algorithm. In the context of Structured Gamma, type checking must ensure that a program maintains the underlying structure defined by a type. It amounts to the proof of an invariant property. We first show that type checking reactions is an undecidable problem if types are defined by arbitrary context-free graph grammars. Then, we propose a sound but incomplete checking algorithm based on the construction of an abstract reduction graph which describes all the possible contexts X for a condition C and type T such that $X + C \overset{*}{\rightarrow}_T \{T\}$. We describe its application to some examples, suggesting that the algorithm is precise enough to tackle most common cases.

In this section and in the appendix, we use C and A to denote the condition and the action of the program or their instantiation to a multiset. The distinction is generally clear from the context.

4.1. The general problem

The reaction $C \models\!\!\Longrightarrow A$ is well-typed for T if

$$\forall M \ (M + C) : T \Rightarrow (M + A) : T$$

Unfortunately, this property is undecidable if types are described by unrestricted context-free graph grammars (such as the structuring types defined in Section 3).

First, let us show that any context-free (word) grammar can be encoded in our formalism. Each letter (terminal) a can be represented by a binary relation $\mathbf{a}$. A word $a_1 \cdots a_n$ is represented by the set $\{\mathbf{a}_1\ x_1\ x_2, \ldots, \mathbf{a}_n\ x_n\ x_{n+1}\}$, that is to say, by the graph

$$\bullet \xrightarrow{a_1} \bullet \cdots \bullet \xrightarrow{a_n} \bullet$$

A context-free (word) grammar in Chomsky normal form can be represented by a context-free graph grammar as follows:

$A \to B\ C$ is represented by $A\ x\ y = B\ x\ z,\ C\ z\ y$

$A \to a$ is represented by $A\ x\ y = \mathbf{a}\ x\ y$

Since the equivalence and inclusion problem of context-free grammars is undecidable, the same results holds in the more general framework of context-free graph grammars.

Let us now consider the following type:

$$\begin{aligned} T &= L\ x \\ L\ x &= \mathbf{f}\ x,\ L_1\ x \\ L\ x &= \mathbf{g}\ x,\ L_2\ x \\ L_1\ x &= \cdots \\ L_2\ x &= \cdots \end{aligned}$$

Suppose that the relations $\mathbf{f}$ and $\mathbf{g}$ do not occur in the definitions of L_1 and L_2. Then, the multisets (or contexts) X such that $|X + \{\mathbf{f}\ x\}| \overset{*}{\rightarrow}_T \{T\}$ are exactly those generated by $L_1\ x$. Similarly, the only possible contexts for $\mathbf{g}\ x$ are those generated by $L_2\ x$. Let us consider the reaction $\mathbf{f}\ x \models\!\!\Longrightarrow \mathbf{g}\ x$. Type checking this reaction with respect to T would prove that $L_1\ x$ generates a language (i.e. a set of multisets) included in the language generated by $L_2\ x$, which is an undecidable problem.

An approach to overcome this theoretical result is to restrict either the type definitions or the form of reactions. There are many sub-classes of context-free grammars that are known to have a decidable equivalence problem. It may then be possible to find a formalism that is powerful enough to describe the most common graph structures

and for which type checking is decidable. In [11], we propose a subclass of types *and* reactions for which a complete (and practical) checking algorithm exists.

We take a different view here, considering checking as an analysis algorithm which is by essence approximate and may declare illegal some valid programs. Most reactions we have encountered so far are within the reach of this algorithm but more experience is needed to decide if the theoretical limitations have a significant impact in practice.

4.2. Overview of the checking process

First, let us note that values and assignments are not relevant for type checking. So, in this section and the following, we consider multisets and rewriting rules restricted to relations. Also, we assume that checking is done relatively to a given type T.

A reduction step by a Structured Gamma program is of the form $M+C \longrightarrow_P M+A$. The algorithm has to check that the application of every reaction of the program leaves the type of the multiset unchanged. In other terms, for any reaction $C \models\!\!\Longrightarrow A$ and multiset $M+C$ of type T, it checks that $M+A$ is of type T (i.e. $M+A \overset{*}{\rightsquigarrow}_T \{T\}$).

The checking algorithm is based on the observation that if $M+C$ has type T, any reduction chain $M+C \overset{*}{\rightsquigarrow}_T \{T\}$ can be reorganized as

$$M+C \overset{*}{\rightsquigarrow}_T X+C \overset{*}{\rightsquigarrow}_T \{T\}$$

where

- no element of C is involved in the reduction chain $M+C \overset{*}{\rightsquigarrow}_T X+C$
- each reduction of $X+C \overset{*}{\rightsquigarrow}_T \{T\}$ involves at least one element of a residual of C. (A residual of C is either C or the result of a rewriting involving one element of a residual of C.)

Such contexts X can be derived from C by considering all the possible reductions of the following form:

$$X_0+C \rightsquigarrow_T C_1 \quad X_1+C_1 \rightsquigarrow_T C_2 \quad \cdots \quad X_n+C_n \rightsquigarrow_T \{T\} \tag{1}$$

Each step is an application of a $\rightsquigarrow_T$ rule involving at least one component of C_i and X_i is a *basic context*. Basic contexts are the smallest (possibly empty) multisets of relations needed to match the *lhs* of a reduction rule. They are therefore completely reduced by the reduction rule. Let $X=X_n+\cdots+X_0$ then

$$X+C=X_n+\cdots+X_0+C \rightsquigarrow_T X_n+\cdots+X_1+C_1 \rightsquigarrow_T \cdots \rightsquigarrow_T \{T\}$$

so $X+C \overset{*}{\rightsquigarrow}_T \{T\}$.

The checking algorithm computes all the possible contexts X for C by considering all possible reductions chains of the form (1). Then, it is sufficient to check the property $X+A \overset{*}{\rightsquigarrow}_T \{T\}$ for all the possible contexts.

Since $M+C$ has type T, there is a least one reduction chain $M+C \overset{*}{\rightsquigarrow}_T \{T\}$, which can be written as

$$M+C \overset{*}{\rightsquigarrow}_T X+C \overset{*}{\rightsquigarrow}_T \{T\}$$

All contexts have been considered and the algorithm has checked that $A+X \overset{*}{\rightsquigarrow}_T \{T\}$, thus

$$M+A \overset{*}{\rightsquigarrow}_T X+A \overset{*}{\rightsquigarrow}_T \{T\}^1$$

and the type of the multiset is maintained (i.e. $M+A$ has type T).

To get round the problem posed by the unbounded length of chains of the form (1), we consider residuals C_i up to renaming of variables. This point is related to the fact that contexts are in general unbounded and, as pointed out in the following section, this forces us to make conservative approximations.

4.3. A checking algorithm

A renaming is a one-to-one mapping and its domain is the set of variables that differ from their image. We will use the following lemma.

Lemma 4. *Let σ be a renaming then $C_1 \rightsquigarrow_T C_2 \Leftrightarrow \sigma C_1 \rightsquigarrow_T \sigma C_2$.*

The type checking algorithm consists in examining in turn each reaction of the program.

$$\text{TypeCheck } (P,T) = \forall (C,A) \text{ of } P.\ \text{Check } (A,T,\ \text{Build } (C,\{C\},T)).$$

For each reaction $C \Longmapsto A$, a reduction graph G, summarizing all possible reduction chains from C to $\{T\}$, is built by Build. Then, Check verifies that for any reduction chain and context X of the graph from C to $\{T\}$, $A+X$ reduces to $\{T\}$. These functions are described in Fig. 2.

Build takes an initial graph made of the root C. The reduction graph is such that nodes are distinct (even up to renaming of variables) residuals C_i and edges are of the form $C_i \xrightarrow{X,\sigma} C_j$. This notation indicates that $C_i+X \rightsquigarrow_T \sigma C_j$ where X is a basic context and σ is a variable renaming. Recall that "$\rightsquigarrow$" reductions have a global condition: variables suppressed by a reduction rule should not occur in the rest of the multiset. To generate valid $\rightsquigarrow$ reduction chains we enforce that variables occurring in a basic context are either variables occurring in the current residual or fresh variables. This condition ensures that we never reintroduce suppressed variables.

---

[1] The global conditions on the reduction $M+A \overset{*}{\rightsquigarrow}_T X+A$ are ensured by the validity of the reduction of $M+C \overset{*}{\rightsquigarrow}_T X+C$ and the fact that variables of A are either variables of C or fresh variables.

Build (C, G, T)
if $C = \{T\}$ **then return** G **else**
let $CX = \{(C_i, X_i) \mid C + X_i \rightsquigarrow_T C_i\}$ **in**
/* *CX is a finite set (up to fresh variable renaming)* */
for each (C_i, X_i) in CX **do**
if $\exists C_j \in G$ and σ_j such that $C_i = \sigma_j C_j$ **then** $G := G + C \xrightarrow{X_i, \sigma_j} C_j$
else $G := G + C_i + C \xrightarrow{X_i, id} C_i$; $G :=$ Build(C_i, G, T)
od
return G

Check (A, T, G)
let $\mathcal{S} = \{(X_0 + \sigma_1 X_1 + \cdots + \sigma_1 \circ \cdots \circ \sigma_n\ X_n,\ \{T\})$
$\mid C \xrightarrow{X_0, \sigma_1} C_1 \xrightarrow{X_1, \sigma_2} \cdots C_n \xrightarrow{X_n, \sigma_{n+1}} \{T\} \in G\}$
and $\mathcal{C} = \{(X_0 + \sigma_1 X_1 + \cdots + \sigma_1 \circ \cdots \circ \sigma_{i-1}\ X_{i-1}, \sigma_1 \circ \cdots \circ \sigma_i\ C_i)$
$\mid C \xrightarrow{X_0, \sigma_1} C_1 \xrightarrow{X_1, \sigma_2} \cdots \xrightarrow{X_{i-1}, \sigma_i} C_i \in G$ and $\exists C_i \xrightarrow{X, \sigma_2} \cdots C_i \in G$
and $\exists C_i \xrightarrow{Y, \sigma_2} \cdots \{T\} \in G$ and $\nexists\ j < i \mid C_j \xrightarrow{Z, \sigma_2} \cdots C_j \in G\}$
in $\forall (X, Y) \in \mathcal{S} \cup \mathcal{C}$. Reduces_to $(A + X, Y, T)$

Reduces_to (X, Y, T)
if X=Y **then** True **else**
if X is irreducible **then** False **else**
let $\{X_1, \cdots, X_n\}$ be the set of all possible residuals of X by a $\rightsquigarrow_T$ reduction
in $\bigvee_{i=1}^{n}$ Reduces_to (X_i, Y, T)

Fig. 2. Type checking functions.

The structure of Build is a depth first traversal of all possible reduction chains. The recursion stops when C is $\{T\}$ or is already present in the graph. CX is the set of basic contexts and residuals denoting all the different reductions of C. Note that basic contexts X_i and residuals C_i may occur several times in CX (there may be several possible reduction rules for the same term and different terms can be reduced in the same residual). However, the set CX is finite since pairs (C_i, X_i) are considered up to renaming of fresh variables introduced by X_i.

If a residual C_i is already present in the graph, that is, there is already a node C_j such that $C_i = \sigma_j C_j$, then the edge $C \xrightarrow{X_i, \sigma_j} C_j$ is added to the graph. Otherwise, C_i becomes a new node and the edge $C \xrightarrow{X_i, id} C_i$ is added.

The function Check takes the graph as argument and performs the following verifications:

- For every simple path (i.e. containing no cycle) from the root to $\{T\}$ with context X, it checks that $A + X \overset{*}{\rightsquigarrow}_T \{T\}$.

  Let us focus on the meaning of a path $C \xrightarrow{X_0, \sigma_1} C_1 \xrightarrow{X_1, \sigma_2} \cdots C_n \xrightarrow{X_n, \sigma_{n+1}} \{T\}$. By definition, we have $C + X_0 \rightsquigarrow_T \sigma_1 C_1, \ldots, C_n + X_n \rightsquigarrow_T \{T\}$ and by Lemma 4 we have

$$C + X_0 + \sigma_1 X_1 + \cdots + \sigma_1 \circ \cdots \circ \sigma_n X_n \overset{*}{\rightsquigarrow}_T \{T\}$$

  So, the context X associated with the above path is $X = X_0 + \sigma_1 X_1 + \cdots + \sigma_1 \circ \cdots \circ \sigma_n X_n$.

- For every simple path with context X from the root to a residual C_i belonging to a cycle, it checks that $A + X \overset{*}{\rightsquigarrow}_T C_i$. In fact, it is sufficient to check this property for the first residual belonging to a cycle occurring on the path from the root and only for cycles from which a path to $\{T\}$ exists.

The verifications that the action A with context X can be reduced to Y are implemented by function Reduces_to$(A + X, Y, T)$. It simply tries all the $\rightsquigarrow_T$ reductions on the term $A + X$ using a depth first strategy. If a path leading to Y is found then True is returned. If Reduces_to finds out that all the normal forms of $A + X$ by "$\rightsquigarrow_T$" are different from Y, it returns False which entails the failure of the verification (TypeCheck$(P, T) =$ False).

The treatment of cycles makes the algorithm incomplete. For each cycle on a node C_i the algorithm enforces that A and the associated context X reduce to C_i. While this is a sufficient condition it is not necessary. More precise solutions exist but they are intricate and, of course, still incomplete.

The termination of TypeCheck is ensured by the following observations:

- The reduction graph is finite.
  - The number of nodes is bounded. Since the *rhs* of the $\rightsquigarrow_T$ rules are always a single element (nonterminal) the number of relations in C_i's never grows. The number of relation names in a type and in a condition C as well as the arity of relations are bounded so the number of different C_i (up to renaming of variables different from $Var(C)$) is bounded.
  - The number of edges is bounded. For any term C_i there is only a finite number of basic contexts matching a $\rightsquigarrow$ rule (up to renaming of fresh variables), and for each basic context there is a finite number of different $\rightsquigarrow$ reductions.
- Reduces_to terminates. It is possible to find a well-founded decreasing ordering for $\rightsquigarrow$ reductions. As usual with context-free grammars, it is always possible to put the type definition in a Chomsky-like normal form such that all $\rightsquigarrow$ rules would be one of the two following forms:

$$NT_1\, x_1 \cdots x_i, \ldots, NT_n\, y_1 \cdots y_j \rightsquigarrow_T NT_m\, z_1 \cdots z_k$$
$$\mathbf{r}\ x_1 \cdots x_i \rightsquigarrow_T NT\, y_1 \cdots y_j$$

Let $nt(T)$ and $nnt(T)$ denote the number of terminals and nonterminals of T respectively, then $T_1 \ll T_2$ iff $nt(T_1) < nt(T_2)$ or $(nt(T_1) = nt(T_2)$ and $nnt(T_1) < nnt(T_2))$ is a well-founded ordering.

The type checking is correct if it ensures that the type of a program is invariant throughout the reduction. The proof amounts to showing a subject reduction property.

Proposition 5. $\forall P, M_1 : T\ M_1 \longrightarrow_P M_2$ *and* $TypeCheck(P, T) \Rightarrow M_2 : T$

The proof can be found in Appendix A.

4.4. Examples

Even if the theoretical complexity of the algorithm is prohibitive, the cost seems reasonable in practice. We take here a few examples to illustrate the type checking process at work.

Example 6. Let us take the *Iota* program working on type *List*.

$$Iota : List = [\textbf{end}\ x,\ \bar{x} > 1 \models\!\!\Rightarrow \textbf{next}\ x\ y,\ \textbf{end}\ y,\ y := \bar{x} - 1]$$

Operations on values are not relevant for type checking and we consider the single reduction rule

$$\textbf{end}\ x \models\!\!\Rightarrow \textbf{next}\ x\ y,\ \textbf{end}\ y$$

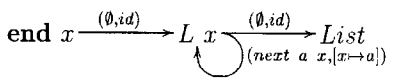
The type definition and the associated $\rightsquigarrow_{List}$ rewriting system are:

$List = L\ x$	$L\ x$	$\rightsquigarrow_{List} List$
$L\ x = \textbf{next}\ x\ y,\ L\ y$	$\textbf{next}\ x\ y,\ L\ y$	$\rightsquigarrow_{List} L\ x$
$L\ x = \textbf{end}\ x$	$\textbf{end}\ x$	$\rightsquigarrow_{List} L\ x$

The type checking amounts to the call

$$Check((\textbf{next}\ x\ y, \textbf{end}\ y), List, Build(\textbf{end}\ x, \{\textbf{end}\ x\}, List))$$

$Build(\textbf{end}\ x, \{\textbf{end}\ x\}, List)$ builds the following reduction graph:



We are left with checking:

- Reduces_to $((\textbf{next}\ x\ y,\ \textbf{end}\ y), List, List)$ which is true because of the following reduction sequence
  $\textbf{next}\ x\ y,\ \textbf{end}\ y \rightsquigarrow_{List} \textbf{next}\ x\ y,\ L\ y \rightsquigarrow_{List} L\ x \rightsquigarrow_{List} List$
- Reduces_to $((\textbf{next}\ x\ y,\ \textbf{end}\ y), L\ x, List)$ which is true because of the following reduction sequence
  $\textbf{next}\ x\ y,\ \textbf{end}\ y \rightsquigarrow_{List} \textbf{next}\ x\ y,\ L\ y \rightsquigarrow_{List} L\ x$

So, $TypeCheck(Iota, List) = True$ and we conclude that the "*List*" invariant is maintained.

Example 7. Let us consider a program performing an insertion at the end of a list with connections to the last element (as defined in Section 3.3).

$$Wrong : ListLast = [\textbf{next}\ x\ z,\ \textbf{last}\ x\ z,\ \textbf{next}\ z\ z \models\!\!\Rightarrow$$
$$\textbf{next}\ x\ z,\ \textbf{next}\ z\ t,\ \textbf{last}\ x\ t,\ \textbf{last}\ z\ t,\ \textbf{next}\ t\ t]$$

Obviously this program is ill-typed. If the list has more than two elements, the first elements would still point to z whereas t is the new last element.

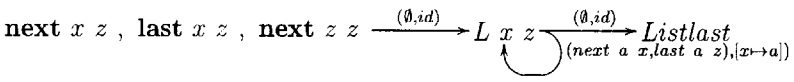
The rewriting system of *Listlast* is

$$L\ x\ z \rightarrow_{Listlast} Listlast$$

$$\textbf{next}\ x\ y,\ \textbf{last}\ x\ z,\ L\ y\ z \rightarrow_{Listlast} L\ x\ z$$

$$\textbf{next}\ x\ z,\ \textbf{last}\ x\ z,\ \textbf{next}\ z\ z \rightarrow_{Listlast} L\ x\ z$$

The reduction graph is



The type checking fails because the action with the empty context does not meet the condition on cycles, namely

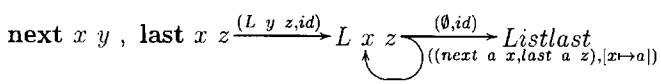
$$\textbf{next}\ x\ z,\ \textbf{next}\ z\ t,\ \textbf{last}\ x\ t,\ \textbf{last}\ z\ t,\ \textbf{next}\ t\ t \not\rightarrow_{Listlast} L\ x\ z$$

and the "*Listlast*" invariant is not maintained.

However, if we consider the insertion program:

$$Add: ListLast = [\textbf{next}\ x\ y,\ \textbf{last}\ x\ z \Longmapsto \textbf{next}\ x\ t,\ \textbf{next}\ t\ y,\ \textbf{last}\ x\ z,\ \textbf{last}\ t\ z]$$

The reduction graph is



It is easy to check that

$$\textbf{next}\ x\ t,\ \textbf{next}\ t\ y,\ \textbf{last}\ x\ z,\ \textbf{last}\ t\ z,\ L\ y\ z \overset{*}{\rightarrow}_{Listlast} L\ x\ z \rightarrow_{Listlast} Listlast$$

and *TypeCheck* yields *True*; the "*Listlast*" invariant is maintained.

5. Refinement of Structured Gamma programs

The introduction put forward two main motivations for the design of Structured Gamma:

- Providing a notation leading to higher-level descriptions of programs manipulating data structures and making it possible to reason about this structure.
- Exposing relevant information to derive more efficient implementations.

The first issue was tackled in the previous sections. Here, we show how Structured Gamma can serve as a basis for program refinements leading to efficient implementations.

The basic source of inefficiency of any "naive" implementation of Gamma is the combinatorial explosion entailed by the semantics of the language for the selection of reacting elements. Let us consider, as an illustration, the following "maximum segment sum" pure Gamma program.

$$maxss = [max_g \circ max_l]$$

$$max_l = [(i,v,s),\ (i',v',s'),\ (i' = i+1),$$

$$(s + v' > s') \qquad \longmapsto (i,v,s),\ (i',v',s+v')]$$

$$max_g = [(i,v,s),\ (i',v',s'),\ (s' \geqslant s) \qquad \longmapsto (i',v',s')]$$

The notation "$\circ$" is used to represent the sequential composition of programs (as in [13, 14]). The input parameter is a sequence of integers. A segment is a subsequence of consecutive elements and the sum of a segment is the sum of its values. The program returns the maximum segment sum of the initial sequence. The elements of the multiset are 3-tuples (i,v,s) where i is the position of value v in the sequence and s is the maximum sum (computed so far) of segments ending at position i. The s field of each 3-tuples is originally set to the v field. The program max_l computes local maxima and max_g returns the global maximum. The complexity (in terms of number of operations) of max_g is linear, even on a naive implementation because any pair of elements (or its mirror) leads to a reaction and the action strictly decreases the size of the multiset. However the worst-case sequential complexity of an unoptimized implementation of max_l is N^3, with N the size of the multiset (i.e. input sequence). This cost is reached by a strategy choosing the first element (i,v,s) in decreasing order of i.

As pointed out in [6, 9], the order in which elements are selected is crucial indeed and most of the refinements leading to efficient optimizations of Gamma programs can be expressed as specific selection orderings. [9] introduces several refinements and shows that they often lead to efficient well-known implementations of the corresponding algorithms. This result is quite satisfactory from a formal point of view because it shows that there is a continuum from specifications written in Gamma to lower-level and efficient program descriptions. These refinements, however, had to be checked manually. Using Structured Gamma as a basis, we can provide general conditions ensuring the correctness of program refinements.

The basic idea, which was already alluded to in Section 3.4, consists in considering multiset (and type) refinements as the addition of extra relations between addresses. These relations are used as further constraints on the control in order to impose a

specific ordering for the selection of elements. We first define the (semantic) notions of refinement on multisets. This notion of refinement is then extended to structuring types and programs.

Definition 8. Let R be a set of relation names, M, and M' multisets, T and T' types and P and P' Structured Gamma programs.

- The restriction of a multiset M' with respect to R is defined as

$$M'\backslash R = M' - \{\mathbf{r}\ a_1 \cdots a_n \mid \mathbf{r} \in R\}.$$

- M' is an R-refinement of M (written $M' >_R M$) iff $M'\backslash R = M$.
- T' is an R-refinement of T (written $T' >_R T$) iff $M' : T' \Rightarrow (M'\backslash R) : T$.
- P' is a partial R-refinement of P (written $P' >_R P$) iff

$$M' \xrightarrow{*}_{P'} N' \Rightarrow M'\backslash R \xrightarrow{*}_{P} N'\backslash R.$$

- P' is a total R-refinement of P (written $P' \gg_R P$) iff

$$M' \overset{*}{\longmapsto}_{P'} N' \Rightarrow M'\backslash R \overset{*}{\longmapsto}_{P} N'\backslash R.$$

The difference between a partial refinement of programs and a total refinement is that only the latter preserves termination.

Let us now illustrate this definition with some examples. The types *Doubly*, *List-last* and $List_m$ defined in Section 3 are refinements of the type *List* (with respect to $\{\mathbf{pred}\}$, $\{\mathbf{last}\}$ and $\{\mathbf{m1}, \mathbf{m2}\}$, respectively), but *Circular* is not a refinement of *List*. As an illustration of the relevance of this definition for deriving efficient implementations of Structured Gamma programs, let us consider yet another refinement of *List*:

$$List_1 = L_1\ x$$

$$L_1\ x = \mathbf{next}\ x\ y,\ \mathbf{i}\ x,\ L_1\ y$$

$$L_1\ x = L_2\ x$$

$$L_2\ x = \mathbf{next}\ x\ y,\ \mathbf{a}\ x,\ L_2\ y$$

$$L_2\ x = \mathbf{end}\ x,\ \mathbf{a}\ x$$

$List_1$ is a list with two extra relations **a** and **i**, which can be seen as "markers" used to distinguish two elements of the list. It should be clear that $List_1$ is a R-refinement of *List* with $R = \{\mathbf{a}, \mathbf{i}\}$. We present now the translation of max_l in Structured Gamma

$$max_l : List = [$$

$$\mathbf{next}\ x\ y,\ (\overline{x}.s + \overline{y}.v > \overline{y}.s) \Longmapsto \mathbf{next}\ x\ y,\ y := (\overline{y}.v, \overline{x}.s + \overline{y}.v)]$$

and a new version max_{l1} which takes advantage of the extra relations to add restrictions on the control:

$$max_{l1} : List_1 = [$$
$$\textbf{next}\ x\ y,\ \textbf{i}\ x,\ \textbf{a}\ y,\ (\overline{x}.s + \overline{y}.v > \overline{y}.s) \Longmapsto \textbf{next}\ x\ y,\ \textbf{i}\ x,\ \textbf{i}\ y,\ y := (\overline{y}.v, \overline{x}.s + \overline{y}.v)$$
$$\textbf{next}\ x\ y,\ \textbf{i}\ x,\ \textbf{a}\ y,\ (\overline{x}.s + \overline{y}.v \leqslant \overline{y}.s) \Longmapsto \textbf{next}\ x\ y,\ \textbf{i}\ x,\ \textbf{i}\ y]$$

In this example, **i** is the relation characterizing inert elements (elements that cannot be modified by a reaction) and **a** corresponds to active elements. It can be shown that max_{l1} is a partial R-refinement of max_l with $R = \{\mathbf{a}, \mathbf{i}\}$. A program P' is a partial refinement of P if P can simulate all the "significant" reactions of P'. The intuition is that the reactions that affect only relations in R are not significant for P. It may be the case however that P' is not a proper implementation of P. The reason is that the termination condition for P' may be "stronger" than the termination condition of P (because of the extra relations). An extra condition has to be imposed to ensure that max_{l1} is a total refinement of max_l. This condition is also expressed in terms of type refinements (roughly speaking, the initial multiset must be of type $List_2 = \textbf{next}\ x\ y,\ \textbf{i}\ x,\ L_2\ y$). The following theorem shows that partial refinement can still serve as the basis of a correct program transformation.

Proposition 9. *If* $P' >_R P$ *then*

$$M' \overset{*}{\longmapsto}_{P'} N' \quad \textit{and} \quad N'\backslash R \overset{*}{\longmapsto}_{P} N \Rightarrow M'\backslash R \overset{*}{\longmapsto}_{P} N$$

The proof of this theorem follows directly from the definition of partial refinement. The interesting consequence is that the property $P' >_R P$ allows us to "replace" P by the sequential composition $P \circ P'$ (with an intermediate conversion of the result N' of P' into $N'\backslash R$).

In the above example, the complexity of the implementation of max_l in Structured Gamma is quadratic provided that the type *List* is implemented in memory as a standard linked list with pointers. So, the translation into Structured Gamma itself leads to a first improvement of the behavior of the program. The complexity of max_{l1} is linear, but it is only a partial refinement of max_l and has to be composed with max_l for the transformation to be correct. If the initial multiset has the correct type $List_2$, then the result of max_{l1} is also a normal form for max_l and the execution of max_l is linear too: it amounts to checking that a stable state has been reached. So partial refinement is strong enough to reduce the complexity to N^3 to $2N$ in this case. Proving total refinement allows us to get rid of max_l and the resulting program is the expected one-pass linear walk through the list.

As a final comment, let us emphasize the fact that simple syntactic criteria can be used to check type and program (partial) refinement. Basically, a type T' is a R-refinement of type T if the definition of T can be obtained (modulo renaming of

nonterminals and cancelling useless rules) by removing from the definition of T' all the occurrences of $(\mathbf{r}\, x_1 \cdots x_n)$ with $\mathbf{r} \in R$. The same idea applies to programs. These purely syntactic criteria can be used to check all the type and program refinements used in this section.

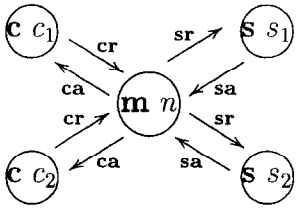
6. Application to coordination

The examples used to illustrate Structured Gamma so far were traditional algorithmic problems. In this section, we take a rather different view at structured multisets which makes Structured Gamma suitable as a coordination language. Coordination languages [5, 15], software architecture languages [12] and configuration languages [18] were proposed as a way to make large applications more manageable and more amenable to formal verifications. They are based on the principle that the definition of a software application should make a clear distinction between individual components and their interaction in the overall software organization. In order to use Structured Gamma as a coordination language, we interpret the addresses in the multisets as names of individual entities to be coordinated. Their associated value defines their behavior (in a given programming language that is independent of the coordination language) and the relations correspond to communication links. A structuring type provides a description of the shape of the overall architecture. As an illustration, a client–server architecture can be specified by a structuring type:

$$
\begin{aligned}
CS &= N\ n \\
N\ n &= \mathbf{cr}\ c\ n,\ \mathbf{ca}\ n\ c,\ \mathbf{c}\ c,\ N\ n \\
N\ n &= \mathbf{sr}\ n\ s,\ \mathbf{sa}\ s\ n,\ \mathbf{s}\ s,\ N\ n \\
N\ n &= \mathbf{m}\ n,\ \mathbf{x}\ x
\end{aligned}
$$

cr c n and **ca** n c denote respectively a communication link from a client c to the manager n (the client request channel), and the dual link from n to c (the client answer channel). The case for servers is similar. Unary relations like **c**, **s**, **m** and **x** characterize the role of an entity (respectively client, server, manager and external entity here). The external entity stands for the external world; it records requests for new clients wanting to be registered in the system.

As an illustration, the following (unconnected) graph represents an instance of a client–server architecture with clients c_1 and c_2 and servers s_1 and s_2. Unary relations are represented by circles and binary relations by arrows.



The architecture can be seen as the skeleton of an application. In order to be executable, it must be 'fleshed', or completed with a mapping from nodes to entities defined in a given language. In [20], we propose a language for programming the individual entities. We provide a structural operational semantics of this language and we show how it cooperates with the semantics of coordination. The specification of the computation of an architecture instance mirrors its hierarchical organization:

- The evolution of the local states of the entities follows the rules of the operational semantics of their programming language.
- The coordinator is in charge of managing the architecture itself (creating and removing entities and links).

Coming back to the client–server architecture, the following two rules specify a correct coordinator:

$$\mathbf{x}\ x,\ \mathbf{m}\ n \quad \Longmapsto \quad \mathbf{x}\ x',\ \mathbf{m}\ n,\ \mathbf{cr}\ c\ n,\ \mathbf{ca}\ n\ c,\ \mathbf{c}\ c$$

$$\mathbf{m}\ n,\ \mathbf{cr}\ c\ n,\ \mathbf{ca}\ n\ c,\ \mathbf{c}\ c \quad \Longmapsto \quad \mathbf{m}\ n$$

The first rule is the addition of a new client and the second one represents the removal of a client from the system. Note that these rules are completed with side conditions on the states of the entities in the complete version of the coordinator presented in [20]; otherwise, the coordinator could clearly lead to infinite behaviors. In the first rule, the side condition bears on the state of the external entity **x** which provides the information for deciding the creation of a new client.

An important advantage of our approach is that coordinators can be checked statically (using the algorithm of Section 4) to ensure that they preserve the style of the architecture. The main departure with respect to previous proposals for the formal definition of software architectures ([1, 16]) is that we consider the overall shape (or geometry) of the architecture as an object of its own. This allows us to check relevant properties of the architecture very easily (for instance, there is no direct communication link between a server and a client in the above architecture). In contrast, [1] uses CSP programs to define the architecture, which leads to a description mixing the communication protocol with the geometry of the communication.

7. Conclusions

Different notions of context-free graph grammars have been studied in the literature. They are defined either in terms of node replacement [10] or in terms of hyper-edge replacement [8]. The graph grammars described in this paper are closely related to Raoult and Voisin's (hyper-)graph rewriting [21]. They define a hyper-graph as a set of hyper-edges, written $fx_1 \cdots x_n$, where f is a function symbol (the counterpart of our relations) and $x_1 \cdots x_n$ are variables denoting vertices (corresponding to addresses in our case). They describe rewriting of sets of hyperedges and provide a criterion for confluence.

The main departure of our work with respect to most of the previous studies of graph rewriting is the fact that we use graphs to represent data structures rather than

programs. The underlying theory is not affected, but this specific point of view entails different kinds of problems (such as the type checking of Section 4).

The types introduced in this paper are context-free graph grammars. This makes the definition of square grids, for example, impossible. It is natural to investigate the extension to types as context-sensitive grammars. With such an extension, a square grid could be described as

$$
\begin{array}{ll}
\mathit{Grid} & = E\ x\ y,\ S\ x\ z,\ L\ y\ z \\
L\ x\ y & = E\ x\ z,\ S\ y\ t,\ L\ z\ t \\
L\ x\ y & = \textbf{end}\ x,\ \textbf{end}\ y \\
E\ x\ y,\ S\ x\ z & = \textbf{east}\ x\ y,\ \textbf{south}\ x\ z,\ E\ z\ t,\ S\ y\ t \\
\textbf{end}\ x,\ E\ x\ y\_1, & = \textbf{end}\ x,\ \textbf{east}\ x\ y,\ \textbf{end}\ z, \\
\textbf{end}\ z, S\ z\ t\_1 & \quad \textbf{south}\ z\ t,\ \textbf{end}\ y,\ \textbf{end}\ t
\end{array}
$$

The semantics of context-sensitive types is defined in the same way as the semantics of context-free types (Section 3). The only difficulty lies in the checking process since context-sensitive $\rightsquigarrow_T$ reductions are not necessarily decreasing with respect to the size of the term. It may be possible to restrict type definitions such that a well-founded order can be found and our checking algorithm adapted. We are currently working on this issue.

We think that the framework developed in this paper can be of interest for applications in various areas. We have already presented program refinement and coordination. Other applications are the specification of networks of processors and the definition of type systems for imperative languages. We just sketch the latter here.

The type systems currently available for imperative languages are too weak to detect a significant class of programming errors. For example, they cannot express the property that a list is doubly-linked or circular. As we have shown in this paper, such structures can be specified naturally using structuring types. We provide in [11] a syntax for a smooth integration of structuring types in C. The programmer can still express pointer manipulations with the expected constant time execution and benefit from the additional guarantee that the property specified by the structuring type is an invariant of the program. The graph types approach [17] shares the same concern. In their framework, a graph is defined using a canonical spanning tree (called the backbone) and auxiliary pointers. Only the backbone can be manipulated by programs and some simple operations may implicitly involve non-constant updates of the auxiliary pointers. In contrast, our types do not privilege any part of the graph and all operations on the structure appear explicitly in the rewrite rules.

Acknowledgements

Thanks are due to the referees who provided helpful suggestions. This work was supported by Esprit Basic Research project 9102 *Coordination*.

Appendix A

Proof of Proposition 5. A rewriting $M_1 \rightarrow_P M_2$ involves a rule $C \Longmapsto A$ and can be described as $M_1 = M + C \rightarrow M + A = M_2$. Since $M + C$ is a multiset of type T then there is a reduction

$$M + C \overset{*}{\rightsquigarrow}_T X_0 + \cdots + X_n + C \overset{*}{\rightsquigarrow}_T \{T\}$$

with the reductions

$$C + X_0 \rightsquigarrow_T C_1,\ C_1 + X_1 \rightsquigarrow_T C_2,\ \ldots,\ C_n + X_n \rightsquigarrow_T \{T\}$$

such as Eq. (1) in Section 4.2.

We consider two cases:

(1) All C_i are different (even up to renaming of variables).

Let us show that the reduction chain considered is represented (up to renaming) in the reduction graph computed by Build$(C, \{C\}, T)$. Note that the type checking algorithm uses two renamings. The first one bounds the number of edges (i.e. makes the set CX finite). We write α_i to denote this kind of renaming (α_i is used implicitly in the definition of CX). The second one bounds the number of nodes; it is denoted by σ_i.

Starting from C, Build considers all the pairs (C', X') such that $C + X' \rightsquigarrow_T C'$ up to renaming of fresh variables introduced by X'. So, it must be the case that a pair $(\alpha_1 X_0, \alpha_1 C_1)$ has been considered in the reduction $C + \alpha_1 X_0 \rightsquigarrow_T \alpha_1 C_1$. But $\alpha_1 C_1$ might have been already present in the graph up to renaming. So, in general, there is an edge $C \overset{X'_0, \sigma_1}{\longrightarrow} C'_1$ in the graph such that $X'_0 = \alpha_1 X_0$ and $\sigma_1 C'_1 = \alpha_1 C_1$.

Using the same reasoning, we are ensured that the graph includes the edges

$$C'_1 \overset{X'_1, \sigma_2}{\longrightarrow} C'_2 \quad \text{with } X'_1 = \alpha_2 \circ \sigma_1^{-1} \circ \alpha_1 X_1$$

$$\ldots$$

$$C'_n \overset{X'_n, \sigma_{n+1}}{\longrightarrow} \{T\} \quad \text{with } X'_n = \alpha_{n+1} \circ \sigma_n^{-1} \circ \cdots \circ \sigma_1^{-1} \circ \alpha_1 X_n$$

Note that the domain of α_i comprises only fresh variables of X_{i-1} and that the domain of σ_j is included in the set of variables of C_j. Thus, if $j < i$ the domains of α_i and σ_j are disjoint and the X'_i can be rewritten as

$$X'_i = \sigma_i^{-1} \circ \cdots \circ \sigma_1^{-1} \circ \alpha_{i+1} \circ \cdots \circ \alpha_1 X_i$$

Now, Check has verified that

$$A + X'_0 + \sigma_1 X'_1 + \cdots + \sigma_1 \circ \cdots \circ \sigma_n X'_n \overset{*}{\rightsquigarrow}_T \{T\}$$

by replacing the X'_i's by their definition, we get

$$A + \alpha_1 X_0 + \alpha_2 \circ \alpha_1 X_1 + \cdots + \alpha_{n+1} \circ \cdots \circ \alpha_1 X_n \overset{*}{\rightsquigarrow}_T \{T\}$$

The domains of α_i's are disjoint (they are renamings of *fresh* variables) so any composition of α_i's can be replaced by a unique variable renaming α whose domain is the set of fresh variables introduced by all the basic contexts. Furthermore, the domain of α is also disjoint from $Var(A)$, hence we have

$$\alpha A + \alpha X_0 + \alpha X_1 + \cdots + \alpha X_n \overset{*}{\rightsquigarrow}_T \alpha\{T\}$$

which implies by Lemma 4

$$A + X_0 + X_1 + \cdots + X_n \overset{*}{\rightsquigarrow}_T \{T\}$$

so $M + A \overset{*}{\rightsquigarrow}_T X_0 + \cdots + X_n + A \overset{*}{\rightsquigarrow}_T \{T\}$.

(2) Otherwise, let C_j, C_k $(j<k)$ be the first residuals such that $\sigma C_j = C_k$.

Using the same reasoning as before we can show that there is a path in the graph

$$C \xrightarrow{X'_0,\sigma_1} C'_1 \cdots \xrightarrow{X'_{j-1},\sigma_j} C'_j \cdots \xrightarrow{X'_{k-1},\sigma_k} C'_k$$

such that

$$X'_i = \sigma_i^{-1} \circ \cdots \circ \sigma_1^{-1} \circ \alpha\ X_i \quad \text{and} \quad C'_i = \sigma_i^{-1} \circ \cdots \circ \sigma_1^{-1} \circ \alpha C_i$$

Since $\sigma C_j = C_k$ and C'_j and C'_k are renamings of, respectively, C_j and C_k, there is a renaming τ such that $\tau C'_j = C'_k$. This corresponds to a cycle in a graph such that C'_j leads to $\{T\}$ and is the first node belonging to a cycle on the path. So, Check has verified that

$$A + X'_0 + \cdots + \sigma_1 \circ \cdots \circ \sigma_{j-1} X'_{j-1} \overset{*}{\rightsquigarrow}_T \sigma_1 \circ \cdots \sigma_j C'_j$$

According to the definition of X'_i and C'_i this can be rewritten as

$$\alpha A + \alpha X_0 + \cdots + \alpha X_{j-1} \overset{*}{\rightsquigarrow}_T \alpha C_j$$

thus, by Lemma 4,

$$A + X_0 + \cdots + X_{j-1} \overset{*}{\rightsquigarrow}_T C_j$$

and, by hypothesis,

$$C_j + X_j + \cdots + X_n \overset{*}{\rightsquigarrow}_T \{T\}$$

so,

$$M + A \overset{*}{\rightsquigarrow}_T X_0 + \cdots + X_n + A \overset{*}{\rightsquigarrow}_T C_j + X_j + \cdots + X_n \overset{*}{\rightsquigarrow}_T \{T\}$$

Therefore, $M + A$ has type T and the subject reduction property holds. □

References

[1] R. Allen, D. Garlan, Formalizing architectural connection, in: Proc. 16th Internat. Conf. Soft. Eng., IEEE Computer Soc. Press, Silver Spring, MD, 1994, pp. 71–80.

[2] J.-P. Banâtre, D. Le Métayer, The GAMMA model and its discipline of programming, Science of Computer Programming 15 (1) (1990) 55–77.

[3] J.-P. Banâtre, D. Le Métayer, Programming by multiset transformation, Comm. ACM 36 (1) (1993) 98–111.

[4] J.-P. Banâtre, D. Le Métayer, Gamma and the chemical reaction model: ten years after, Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.

[5] N. Carriero, D. Gelernter, Linda in context, Comm. ACM 32 (4) (1989) 444–458.

[6] M. Chaudron, E. de Jong, Towards a compositional method for coordinating Gamma programs, in: Proc. Coordination'96, Conf., Cesena, Lecture Notes in Computer Science, Vol. 1061, Springer, Berlin, 1996, pp. 107–123.

[7] D. Cohen, J. Muylaert-Filho, Introducing a calculus for higher-order multiset programming, in: Proc. Coordination'96 Conf., Cesena, Lecture Notes in Computer Science, Vol. 1061, Springer, Berlin, 1996, pp. 124–141.

[8] B. Courcelle, Graph rewriting: an algebraic and logic approach, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990, Chapter 5.

[9] C. Creveuil, Techniques d'analyse et de mise en œuvre des programmes Gamma, Ph.D. Thesis, University of Rennes, 1991.

[10] P. Della Vigna, C. Ghezzi, Context-free graph grammars, Inform. and Control 37 (1978) 207–233.

[11] P. Fradet, D. Le Métayer, Shape types, in: Proc. Principles of Programming Languages, ACM Press, Paris, 1997, pp. 27–39.

[12] D. Garlan, D. Perry, Editor's Introduction, IEEE Trans. Software Engineering, Special Issue on Software Architectures, 1995.

[13] C. Hankin, D. Le Métayer, D. Sands, A calculus of Gamma programs, in: Proc. 5th workshop on Languages and Compilers for Parallel Computing, Yale, Lecture Notes in Computer Science, Vol. 757, Springer, Berlin, 1992, pp. 342–355.

[14] C. Hankin, D. Le Métayer, D. Sands, A parallel programming style and its algebra of programs, in: Proc. PARLE Conf., Munich, Lecture Notes in Computer Science, Vol. 694, Springer, Berlin, 1993, pp. 367–378.

[15] A.A. Holzbacher, A software environment for concurrent coordinated programming, Proc. 1st Internat. Conf. on Coordination Models, Languages and Applications, Lecture Notes in Computer Science, Vol. 1061, Springer, Berlin, 1996, pp. 249–266.

[16] P. Inverardi, A. Wolf, Formal specification and analysis of software architectures using the chemical abstract machine model, IEEE Trans. on Software Engrg. 21 (4) (1995) 373–386.

[17] N. Klarlund, M. Schwartzbach, Graph types, in: Proc. Principles of Programming Languages, ACM Press, New York, 1993, pp. 196–205.

[18] J. Kramer, Configuration programming. A framework for the development of distributable systems, Proc. COMPEURO'90, IEEE Press, New York, 1990, pp. 374–384.

[19] D. Le Métayer, Higher-order multiset programming, in: Proc. DIMACS Workshop on Specifications of Parallel Algorithms, Dimacs series in Discrete Mathematics, Vol. 18, American Mathematical Society, Providence, RI, 1994.

[20] D. Le Métayer, Software architecture styles as graph grammars, in: Proc. ACM SIGSOFT'96 4th Symp. on the Foundations of Software Engineering, 1996, pp. 15–23.

[21] J.-C. Raoult, F. Voisin, Set-theoretic graph rewriting, in: Proc. Internat. Workshop on Graph Transformations in Computer Science, Lecture Notes in Computer Science, Vol. 776, Springer, Berlin, 1993, pp. 312–325.